ANN. MULT. GPU PROG.

Implementation of the Pipeline Parallel Programming Technique as an HLPC: Usage, Usefulness and Performance

Mario Rossainz-López[1*], Manuel I. Capel[2], Odon D. Carrasco-Limón[1], Fernando Hernández-Polo[1], Bárbara E. Sánchez-Rinza[1]

*Correspondence:
rossainz@cs.buap.mx

[1]Benemérita Autonomous University of Puebla. Avda. San Claudio and 14 South, San Manuel. 72000, Puebla, Puebla, Mexico.

[2]Department of Software Engineering, College of Computing and Telecommunications. ETSIIT Daniel Saucedo Aranda s/n, 18071 Granada, Spain.

Abstract

This article presents the pipeline communication/interaction pattern for concurrent, parallel and distributed systems as a high-level parallel composition (HLPC) and discusses its usefulness for deriving parallel versions of sequential algorithms. In particular, we provide examples of the parallel solution for the following problems: adding numbers, sorting numbers and solving a system of linear equations. An approach based on structured parallelism and the parallel object concept is used to solve these problems. In its generic pattern, the pipeline pattern is shown as an HLPC that deploys three types of parallel objects (a manager, various stages and a collector) which are interconnected to form the pipeline processing structure. We also use a method to systematically create the HLPC pipeline and solve this type of problem. Each pipeline instance must be able to handle predefined synchronization restrictions between processes (maximum parallelism, mutual exclusion and synchronization of the producer-consumer type, the use of synchronous, asynchronous and future asynchronous communication, etc.). Finally, the article presents the performance of pipeline HLPC-based implementations of parallel algorithms for solving the problems raised in the paper by using exclusive CPUs.

Keywords: HLPC Pipeline, Structured Parallelism, Parallel Objects, Pipeline, Functional Decomposition, Object-oriented Programming

1. Introduction

Among the plethora of object-oriented systems, concurrent object-based programming frameworks are currently only known by the scientific community involved in studying parallelism. There are, however, also notations that allow complete expressions of the behavior of parallel programs intended for specific OO structured parallel programming environments [1] to be written. One first approach that attempts to tackle the problem of

algorithm and program parallelization consists in trying to automatically produce a parallel version of a sequential code with the help of a specific parallelizing environment. A promising alternative approach is the one adopted in this research and this entails following the general structured parallelism method based on a predefined construction known as a high-level parallel composition.

Parallel applications generally follow predetermined execution patterns which are seldom arbitrary or unstructured in their logic [2]. Parallel high-level compositions (HLPC, or CPAN in Spanish) are well-defined and logically well-structured, parallel communication patterns which, once their components and communication scheme have been selected, can be integrated as additional programming constructs to an object-oriented programming language and made available as high-level abstractions in user applications.

The structured parallel programming approach is based on the use of communication/interaction patterns (pipelines, farms, trees, etc.) between the processes and these are usually defined during the design phase of a user application. By abstracting the interaction pattern between concurrent processes, our approach enables applications to be designed in terms of HLPCs which are capable of implementing such patterns. Encapsulation of an HLPC must follow the modularity principle so that the objects and their parallel behavior can be effectively reused to solve problems when the software is implemented. When this is achieved, we can say that a generic parallel pattern has been created to represent process interaction in programs and user applications where it is deployed, independently of the functionality of such processes. This proposal also concurs with the structured parallel programming approach that we have considered and which enriches the modeling capacity of traditional parallel environments by generating libraries of program skeletons [3] that represent specific communication patterns between concurrent processes. Accordingly, rather than programming an application from scratch, we can simply identify the HLPCs which implement suitable communication patterns and use these with the sequential code which has been individually programmed by the application processes. We have identified several parallel patterns of use for implementing significant interconnections and which are reusable in multiple applications and parallel algorithms. The set of patterns found has resulted in a wide library of communication patterns between concurrent processes such as the HLPCs detailed in [4, 5].

During our research, we implemented the pipeline pattern as a generic HLPC. Thanks to the object-oriented programming paradigm, we have parallelly implemented three applications (number addition, number sorting and solving a system of linear equations) by using three different HLPC-based strategies of parallel implementation from the corresponding sequential algorithms. In this way, the software requirements are those that direct the actual semantics of the instance of the HLPC pipeline to be deployed in the user application. Finally, we analyze the performance of the presented applications with Amdahl's Law Speed-up in terms of the implementation of the HLPC pipeline instance and the number of reserved processors in a parallel computer.

2. The Pipeline Technique

This parallel processing technique is applicable to a wide range of problems that are naturally partially sequential, i.e. we can use this technique to solve a problem by splitting it into a series of successive tasks so that data flow in the direction given by the process interconnection structure. Each task can therefore be completed sequentially [6]. In a pipeline, each task is executed by a processor or process as shown in Figure 1. Each process or processor that comprises a pipeline is usually called a stage [7].

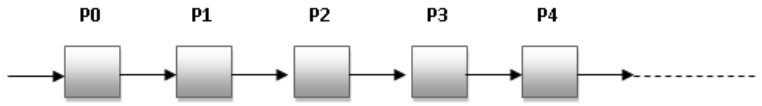

Figure 1. Pipeline structure

Each pipeline stage helps solve the problem as a whole and passes necessary information on to the next stage in the sequence. This type of parallelism is seen as a form of functional decomposition or segmented computing since the problem is divided into separate functions that can be executed individually and independently. This technique assumes that the application functions are executed in succession [6, 7]. An algorithm that solves a certain problem can be formulated as a pipeline if it can be divided into a series of functions that could be executed by the pipeline stages. The general functionality of any pipeline stage (understood as an operation of the program) is shown by the following algorithm:

```
Algorithm StagePipe. Code to be run at any pipeline stage
         {
      ENTRY:  int    j; // current stage index or identifier
                        // Stage 0 receives a set of items
      EXIT: returns the set of items processed by stage j of
the pipeline
      For i = 1 until m do
             {
      receives (item, j-1);
      process (item, j);
      send (item, processed, j + 1);
              }
       }
```

Each pipeline stage must compute a set of items that, in order to be processed, require the information produced by the previous pipeline stage. Once the item has been processed, it must be sent to the next pipeline stage. For reasons of simplicity, the algorithm assumes that each stage computes "m" items at the same time and that each item is at a different stage of the pipeline. The exceptions to this are the first stage of the pipeline where nothing is received from a previous stage and the final stage where nothing is sent to a subsequent stage. One example of a sequential program that can be formulated as a pipeline is a simple cycle or loop in which all the elements of an array are added into an accumulated sum [8].

```
for i = 0 until (n-1) do
{
    sum = sum + a[i];
}
```

This cycle can be viewed separately, instruction by instruction, as the following code shows:

```
sum = sum + a[0];
sum = sum + a[1];
sum = sum + a[2];
sum = sum + a[3];
sum = sum + a[4];
```

One solution using the pipeline might be to assign each sentence to a different stage as shown in Figure 2 and to apply the previous Stage pipeline algorithm [8].

Figure 2 also shows how each stage receives the cumulative sum as its input Sin and an element a[] as its input a to produce the new accumulated sum which is then sent through its output Sout. Therefore, stage i performs the operation Sout = Sin + a [i].

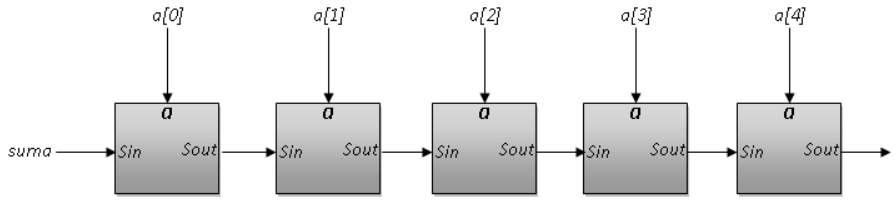

Figure 2. Representation of a loop using a pipeline

Instead of simple operations, however, a series of functions could be assigned to the pipeline stages by applying a functional decomposition strategy [7, 8]. Let us suppose, for example, that we want to sort a set of unsorted data in descending order but we have no sorting algorithm to hand to obtain an array with the data placed from lowest to highest in ascending order. If this sorting algorithm is used, it is necessary to invert the sequence of the data and then sort them. With the pipeline, however, this can be carried out by programming an additional pipeline with an assigned function to invert the output sequence of the first pipeline and then carry out the required processing (Figure 3).

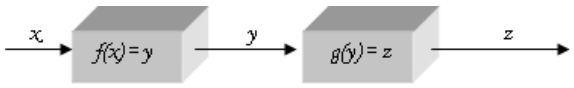

Figure 3. Pipeline with functional decomposition

The elements in the previous figure can be interpreted as follows:

- x represents the initial data set that we assume will be in disarray;
- f (x) represents the function "orders" that receives as input the set of data to be sorted and provides as output the order in ascending order of the data set received;
- y represents the output of the function f (x), i.e. the ordered data;
- g (y) represents the "inverted" function that receives the result of the function "orders" to output the set of previously ordered data but inverted in sequence in descending order;
- Assuming that the input data in this example are integers, the sequence of results in the pipeline is shown in Figure 4.

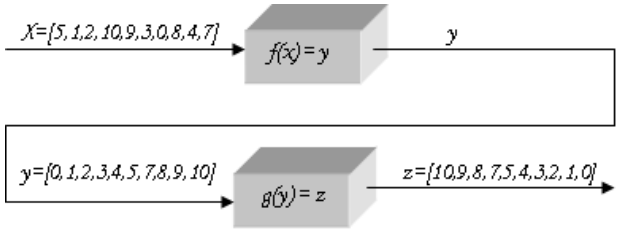

Figure 4. Sequence of results in a pipeline with functional decomposition

If a problem can be divided into a series of sequential tasks, the pipeline approach can therefore yield an increase in execution processing speed according to the following three calculation types proposed in [8]:

1. TYPE A: when more than one instance of the problem solution can be run in parallel. Figure 5 depicts a space-time diagram that shows a possible use of the pipeline for speeding up this type of calculation. The diagram assumes that all processes have the same execution time to complete their task. Each time period is called a pipeline cycle. Each instance of the solution to this problem therefore requires 6 sequential processes (P0 to P5) to generate a ladder effect, each of which completes an instance of the problem in every pipeline cycle. With p-processes (stages) of the pipeline and m-instances of the problem, the number of pipeline cycles required to execute the m-instances is m + p-1 cycles.

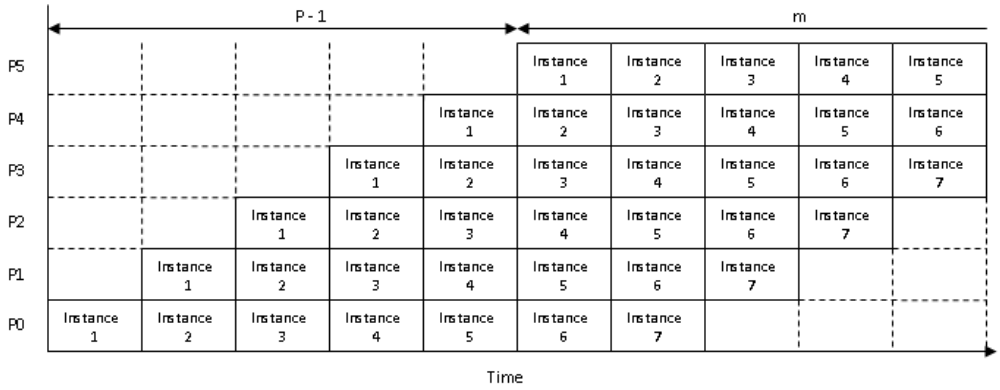

Figure 5. Space-Time diagram of a pipeline

2. TYPE B: when a series of data can be processed and each is used in multiple operations. This appears in arithmetic calculations where a series of data is processed in sequence, such as multiplying elements in a matrix. In this calculation, the individual elements enter the pipeline as a sequential series of numbers. This type of calculation is illustrated in Figure 6, and as an example, there are 10 processes (stages) of the pipeline and 10 elements d0 to d9 that need to be processed.

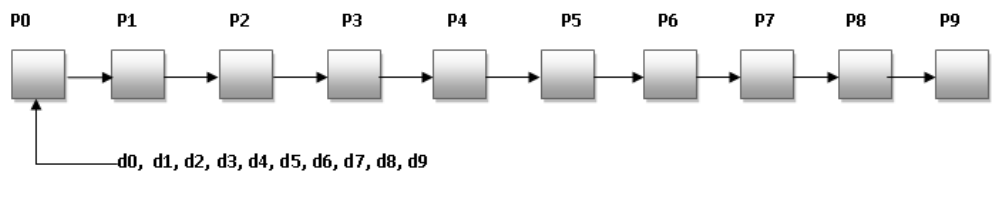

Figure 6. Pipeline for arithmetic calculations

3. TYPE C: when the information required by the next process in the pipeline to start its calculation is passed before the current process has completed all of its internal operations, and so it cannot produce the results and pass them on. This type of calculation is used in parallel programs where there is only one instance of the solution to the problem to be calculated, but each process (stage) can pass information to the next which can then complete its assigned task. Figure 7 shows the

space-time diagrams when information is passed in the pipeline before execution of any of the processes shown has been completed.

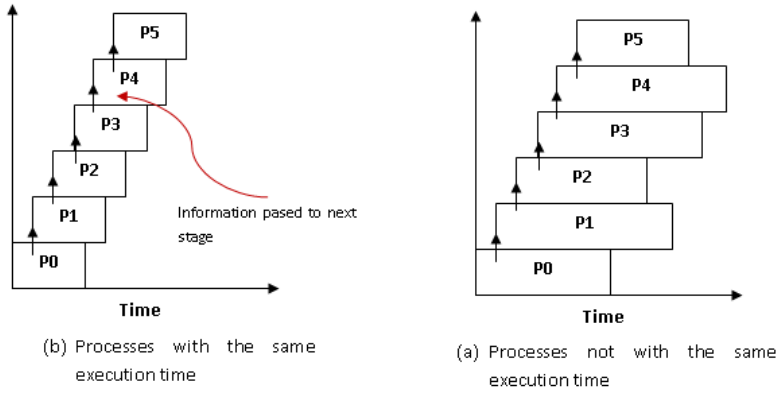

Figure 7. Processing of the pipeline where the information passes from one stage to another before execution of any of the stages has been completed

3. High-level Parallel Compositions (HLPC)

An HLPC comprises three types of parallel objects: firstly, a manager object that represents the HLPC to make an encapsulated abstraction to hide its internal structure; secondly, stage objects for client/server interface encapsulation; and thirdly a collector object to receive the results from the stages. The manager-object controls the references of the other objects in the HLPC and coordinates their execution which is carried out in parallel. Stage objects are responsible for encapsulating a client-server type interface that is established between the manager and a group of passive objects (called slaves) containing the algorithm that solves the sequential problem. The Collector object parallelizes the input of results from the stages and stores them. Manager, Collector and Stages are parallel objects (POR) with their own execution capacity [9].

Applications programmed according to the parallel object (PO) model can exploit both inter-object parallelism and intra-object or internal parallelism. An application based on the PO paradigm has a similar structure to that of Smalltalk objects although it also includes a previously established scheduling policy that specifies the synchronization of operations of one or more objects that can be invoked in parallel [9, 10]. In the PO paradigm, parallel objects support multiple inheritance and this enables a completely new PO specification to be derived.

The following properties of an HLPC were studied in detail in [11]: synchronous, asynchronous and future asynchronous communication between the parallel HLPC objects; internal parallelism of objects, which includes the availability of synchronization mechanisms; parallel mechanism type includes maximum parallelism, mutual-exclusion intended constructs, and producer-consumer concurrency; generic type control features; transparency in the distribution of parallel applications; and programmability-portability-performance of code and applications.

In the HLPC version of the PO programming model, the following basic classes are needed to define the manager, collector, stages objects of an HLPC: an instance of a specific class derived from the ComponentManager class (called manager) to represent an HLPC within an application programmed according to the PO model; instances (called stages) of a specific class derived from the ComponentStage class which are interconnected to implement a series of stages and each stage directs the parallel execution of a slave object which is controlled by the stage itself.

It is also necessary to mention that the creation of stages, collectors and their subsequent interaction is handled transparently to the application user code by the manager. When users want to use an HLPC within an application, not only must they create an instance of a particular manager class, i.e. one that implements the parallel behavior required by the application and to initialize it with the appropriate reference to the stage objects, which in turn control the slave objects, but also declare the name of the requested method (see Figure 8, [11] for more details). From the point of view of reusing the parallel behavior already defined in an HLPC, the most relevant class to instantiate will therefore be that of the manager.

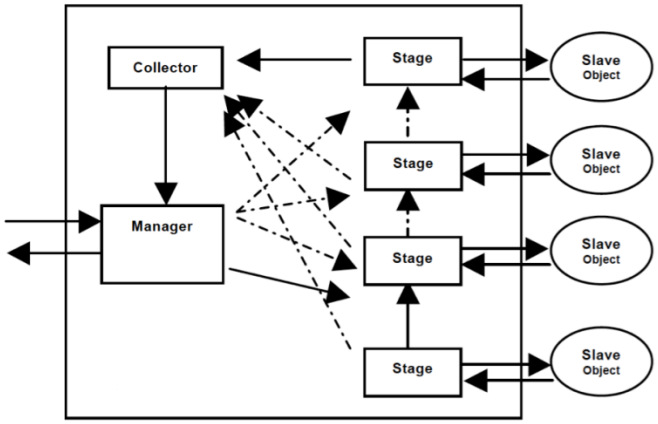

Figure 8. Internal structure of a High-level Parallel Composition or HLPC

3.1. The HLPC Pipeline

The parallel pipeline processing technique is presented as a high-level parallel composition for solving a wide variety of problems from a number of different areas that can already be solved by partially sequential algorithms. In this way, the HLPC pipeline guarantees code parallelization of the resulting algorithm while taking advantage of existing sequential algorithms by using the HLPC processing pattern.

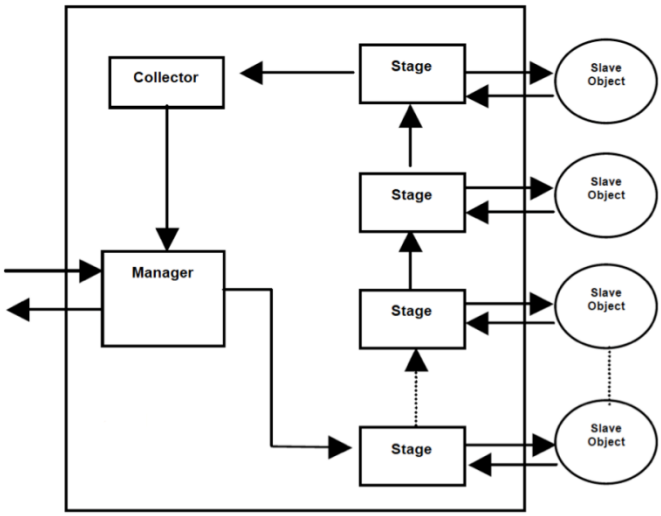

Figure 9. The Pipeline as an HLPC

The *Stage_i and Manager* Objects in Figure 9 are instances of specific classes which inherit the characteristics of *ComponentManager* and *ComponentStage* baseline classes, respectively. These classes redefine the abstract methods of their superclasses. The *Collector* object is the only one that will be an instance of the *ComponentCollector* singleton class. Once the objects have been created and properly connected according to the Pipeline parallel pattern, an HLPC is obtained as an instance of a specific type of parallel pattern after allocating objects associated with the slave stages. The HLPC pipeline is described by the *PipeManager* class, which inherits from the *ComponentManager* class and implements a pipeline communication pattern where its stages are instances of the *PipeStage* class and inherits from the *ComponentStage* class. Any *PipeManager* object can only interact with the first stage, some stages or all stages of the pipeline at the time of initialization according to the pipeline model that we want to implement as an HLPC.

During the execution of a service invocation, the first stage is the only one commanded to create and start the next stage, and so forth. Each PipeStage object creates the following stage of the pipeline during its initialization phase. During the execution phase, an object stage only directly interacts with the next one on the pipeline or with the Collector (or both). Finally, the last stage sends the result to the *Collector* object (instance of the *ComponentCollector* class) and the reference is usually transmitted dynamically through the stages although this will depend on the pipeline model that we want to implement as an HLPC.

3.2. Usage methodology of the HLPC Pipeline

1. The slave objects, which are executed by the stages and represent instances of the functionality required from the HLPC pipeline, are created.
2. The sequential algorithm that solves the initial problem is implemented and its replicas (or instances) become methods associated to the slave objects.
3. A list of associations (e.g. slave object, associated method) are created.
4. An instance of the PipeManager class is created to represent the HLPC pipeline manager being constructed. It is then initialized with the list of associations from the previous step. At this moment, the necessary internal stages of the PipeManager HLPC are automatically created as instances of the PipeStage class and each obtains an association.
5. The initial data to be processed are specified by creating data types and user-defined data as objects, whereas the input data set represents the problem to solve.
6. Application execution is requested to find a solution to the problem.

   Parallel execution is achieved by transparently using synchronous, asynchronous and future asynchronous communication for the user's main program. The HLPC pipeline then creates its *Collector* object, which is passed as a reference to every stage connected with the *Collector* object, together with a copy of the data set to be processed. The initial stage works with these data and they are processed by the slave object associated to each stage. The result is then passed on to the next stage which is then created and initialized by the previous stage. In the same way, the second stage processes the data received when its slave object method is executed and the new result is sent to the next stage, and so on. Finally, the obtained results are received by the *Collector* object which then compiles, processes and sends them to the *PipeManager* object which, in turn, sends the solution outside the HLPC.

7. Finally, the final results are shown to the user.

4. Usage and usefulness of the HLPC Pipeline

A possible HLPC pipeline for solving a variety of problems using this parallel processing pattern is shown below.

4.1. Adding numbers

Let us now consider the problem of adding a series of numbers. A solution can be designed by using the HLPC pipeline so that each stage-object adds the number input from its previous stage to an internally accumulated sum before being sent to the next stage. In order to do so, we will consider a *producer-consumer* communication type, as suggested in [12]. The input data, i.e. the numbers to be added, are sent by the HLPC pipeline manager object in parallel to the stages (one number to each stage) as shown in Figure 10. Every stage then executes the solution-algorithm which is contained in the slave object associated to each stage. The final result of this calculation is sent by the final stage to the collector object, which in turn sends it to the manager which returns the final result to the user. Execution of this HLPC pipeline can clearly be classified as a TYPE A pipeline calculation (see Section 2), i.e. each stage of the HLPC pipeline carries out similar actions in each pipeline's cycle.

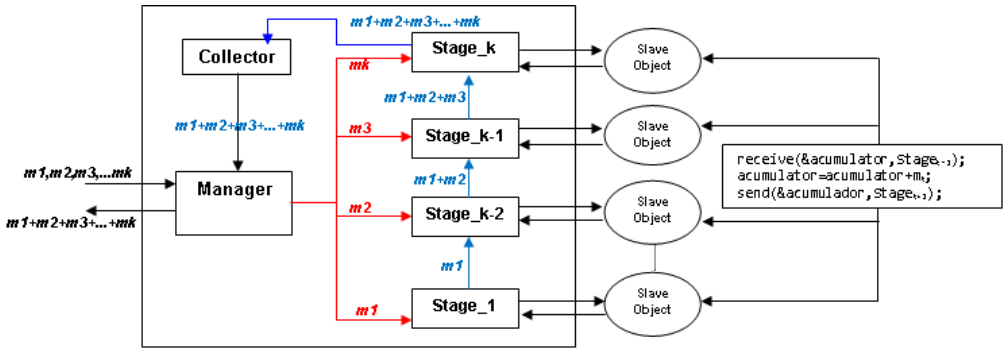

Figure 10. Addition using the HLPC pipeline

4.2. Sorting numbers

In the operation of a parallel sorting algorithm based on the pipeline parallel processing pattern, three phases can be distinguished as indicated in [12, 13]:

- Initial data loading: all the data are delivered to each process associated with two of the pipeline stages. In this phase, the processes are usually running the same code as in the second phase, the only difference being that the processes must be properly initialized in order to receive the first data coming from the previous stage or the data initially loaded into the program.
- Processing the data stream with maximum efficiency: processes behave cyclically during execution. They input and process data from the previous stage and the result is sent to the next stage. Each process must be synchronized with the previous and following stage so that new data are not sent until all the current data have been processed. The final pipeline process behaves differently to the processes of previous stages since it must execute a routine or exit code and send results.
- Downloading: in the final stage of the pipeline, processes send the results of the data that were last processed in the pipeline. The final stage processes will no longer receive data from the input stream and must detect the state of termination. In order for the processes to transmit the data stored in their stages before they finish, a special value is usually introduced at the end of the input sequence used to download the pipeline.

The implementation of the parallel sorting algorithm consists of a process pipeline that is fed with an unsorted sequence of integers by a routine or input code. As a result, the pipeline sort algorithm obtains the sorted integer sequence in ascending order. The number of values of the input sequence cannot be greater than the

number of pipeline stages since each pipeline process can only store one integer, which will be the largest one received so far from the previous stage. At the end of the computation, each stage therefore stores one number of the input sequence in ascending order [12,13]. This represents the parallel sorting algorithm that has been implemented as an HLPC pipeline and is shown in Figure 11.

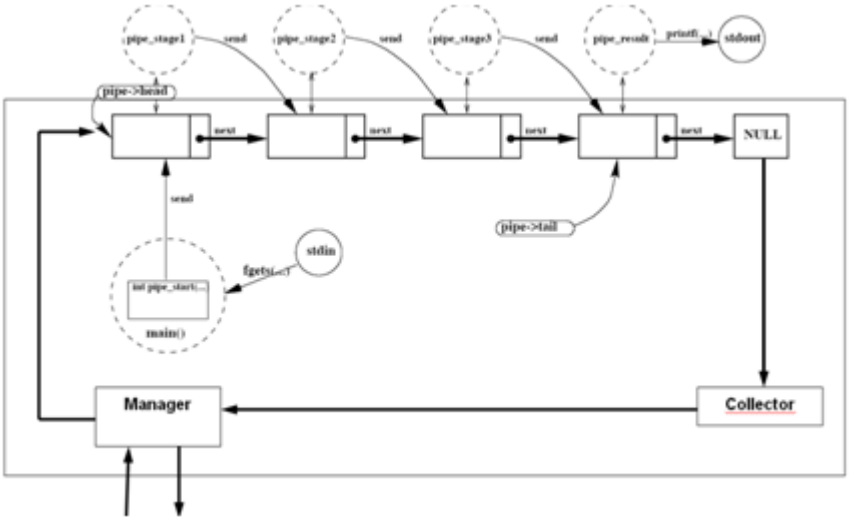

Figure 10. Sorting numbers using the HLPC pipeline

The previous implementation of the pipeline as an HLPC can be classified as a TYPE B pipeline calculation (see Section 2), i.e. the series of numbers to be sorted are processed in sequence. In this calculation, each number enters the HLPC pipeline as a sequential series of numbers.

4.3. Solving a system of linear equations

In this case, the HLPC implementation of the solution can be considered as a TYPE C pipeline calculation (see Section 2) since the stages that comprise the HLPC pipeline continue to work even after information is passed to the next stage. The objective in this case is to solve a system of linear equations such as the following one:

$$
\begin{aligned}
&a_{n-1,0}X_0+a_{n-1,1}X_1+a_{n-1,2}X_2+\ \cdot\ \cdot\ \cdot+a_{n-1,n-1}X_{n-1} = b_{n-1}\\
&\cdot\\
&\cdot\\
&a_{2,0}X_0+a_{2,1}X_1+a_{2,2}X_2 = b_2\\
&a_{1,0}X_0+a_{1,1}X_1 = b_1\\
&a_{0,0}X_0 = b_0
\end{aligned}
$$

The method used to solve this system of linear equations and find the values of X0, X1, X2,. . ., Xn-1, is a simple backward substitution which is repeated until the unknowns of the original system of equations have been determined, as detailed in [12,13]. We first find the value of X0 of the last equation of the system:

$$X_0 = b_0/a_{0,0}$$

The value obtained for X0 is substituted in the following equation of the system, to obtain X1:

$$X_1 = (b_1 - a_{1,0}X_0)/a_{1,1}$$

The values obtained from X1 and X0 are replaced in the third equation of the system, in ascending order, to obtain X2:

$$X_2 = (b_2 - a_{2,0}X_0 - a_{2,1}X_1)/a_{2,2}$$

This is repeated until all the unknown variables have been found. This method of solving a system of linear equations can therefore be implemented using the HLPC pipeline shown in Figure 12.

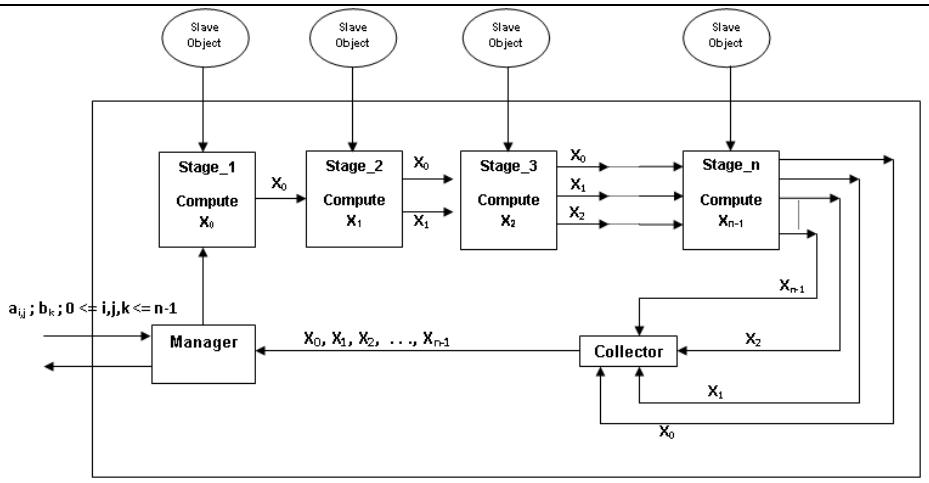

Figure 12. Solving a system of linear equations using the HLPC pipeline

The first stage of the HLPC pipeline calculates and obtains X0, and then sends the result to the second stage which calculates and obtains X1 by using X0 in its calculations. The final stage sends the values of X0 and X1 to the next stage, which calculates X2 using the values of X0 and X1, and so on. The slave objects are associated with the routine that finds the value of Xi from the values of X1, X2, X3….Xi-1 received from the previous stages in order to perform the calculations.

5. Performance of the HLPC Pipeline

The performance analysis of adding numbers, sorting numbers and solving a system of linear equations implemented by the HLPC pipeline was conducted on a parallel computer with 64 processors, 8GB of main memory, high-speed buses with a generic, distributed-shared-memory architecture. Performance of the parallel algorithms implemented using the HLPC pipeline (which deployed the three types of pipeline-based calculations as discussed in Section 2) was measured according to the following execution restrictions:

- For the number addition problem, the same sequential algorithm is used in each of the slave objects associated with the pipeline stages by implementing a TYPE A pipeline calculation as described in Section 4.1. In this case, 50,000 integer numbers (randomly obtained in a range from 0 to 50,000) were input into the pipeline.
- For the sorting number problem, the same sequential routine was used to compare values in each of the slave objects associated with all the pipeline stages (except the first and last ones) as discussed in Section 4.2. The series of calculations performed in the pipeline for HLPC implementation of the solution corresponds to a TYPE B pipeline calculation. For this study sub-case, 50,000 integers were randomly generated and sorted.

- For the problem of the system of linear equations, we worked with a system of 50,000 linear equations. Each equation contained 1 to 50,000 terms, which involved generating random integers to assign the coefficients ai and the independent terms bi, in a range from 1 to 50,000. In the same way as in the previously solved examples, the same routine was used for each of the slave objects associated with the pipeline stages to find the values of the unknown variables Xi of the system and so a TYPE C pipeline calculation was implemented with the HLPC pipeline. These parallel program execution conditions enable us to assume a load which is sufficient for the processors and to show the improvement in HLPC pipeline performance in solving the problems discussed in this paper. For every problem, the entire computation was carried out with 2, 4, 8, 16 and 32 exclusive processors and the results are shown in Tables 1, 2 and 3 and in the graph in Figure 13.

Execution of HLPC Pipeline in exclusive CPUs for the solution of adding of 50000 random numbers						
	HLPC PIPE SEQ.	**CPUSET 2**	**CPUSET 4**	**CPUSET 8**	**CPUSET 16**	**CPUSET 32**
Run time in seconds	199.87	86.62	76	63.7	58.5	49.1
SpeedUp	10.51	2.31	2.63	3.14	3.42	4.07
Amdahl	1.00	1.82	3.08	4.71	6.40	7.80

Table 1. HLPC pipeline performance in adding 50,000 integers

Execution of HLPC Pipeline in exclusive CPUs for the solution of sorting of 50000 random numbers						
	HLPC PIPE SEQ.	**CPUSET 2**	**CPUSET 4**	**CPUSET 8**	**CPUSET 16**	**CPUSET 32**
Run time in seconds	238.5	127.27	123.83	116.97	115.63	111.03
SpeedUp	8.81	1.87	1.93	2.04	2.06	2.15
Amdahl	1.00	1.89	3.39	5.63	8.42	11.19

Table 2. HLPC pipeline performance in sorting 50,000 integers

Execution of HLPC Pipeline in exclusive CPUs for the solution of a system of 50000 linear equations						
	HLPC PIPE SEQ.	**CPUSET 2**	**CPUSET 4**	**CPUSET 8**	**CPUSET 16**	**CPUSET 32**
Run time in seconds	2100.12	1388.1	1351.26	1099.34	1056.94	1031.84
SpeedUp	1.00	1.51	1.55	1.91	1.99	2.04
Amdahl	1.00	1.67	2.50	3.33	4.00	4.44

Table 3. HLPC pipeline performance in solving a system of linear equations

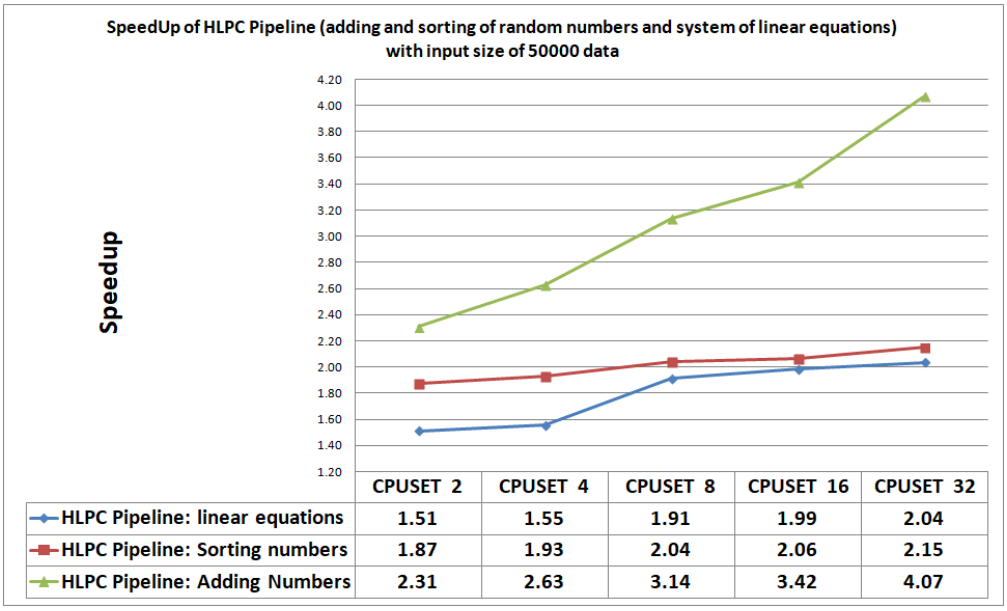

Figure 13. Scalability of the Speedup found for the HLPC pipeline in solving the problems of adding numbers, sorting numbers and the linear equation system

6. Conclusions

This article introduces a new method for designing software applications which is based on the construction of high-level parallel compositions or HLPCs. The method can be used for software development on different platforms, one of the most widely used being C ++ and POSIX Threads [14] on which the programs that support this study were developed. The HLPC pipeline was implemented to provide potential users of this service with an ample library of classes based on the parallel object programming paradigm. In addition to HLPC pipeline implementation, different communication patterns such as farms, trees, etc. were also made available. We showed the purpose and use of HLPC pipeline from a practical point of view by presenting it as a communication pattern between concurrent processes that can be easily used by programmers with little parallel programming experience. The HLPC pipeline has been reused in the communication/interaction between the parallel processes in three examples (adding numbers, sorting numbers and linear equation system) and these have been solved by using three different types of pipeline computations. It has been possible to reuse the application software by adopting the PO approach and this proved to be useful for defining new specific patterns based on previously identified and programmed [15] ones. We were able to obtain efficient code from high abstraction level programs simply by programming the sequential parts of diverse applications and the results were tested in terms of predicted speed-up using Amdahl's Law for a restricted range of 2, 4, 8, 16 and 32 exclusive processors for these executions. In the future, we will analyze the suitability of using the HLPC pipeline to parallelize different algorithms and application programs. We will also investigate its adaptation to a particular case study of application to the problem of the automation of DNA annotation sequences in genome construction.

References

1. Corradi A, Leonardo L, Zambonelli F.: Experiences toward an Object-oriented Approach to Structured Parallel Programming. DEIS technical report no. DEIS-LIA-95-007. (1995).

2. Brinch Hansen: Model Programs for Computational Science: A programming methodology for multicomputers. Concurrency: Practice and Experience, Vol. 5, No. 5, Pp. 407-423. (1993).
3. Darlington et al.: Parallel Programming Using Skeleton Functions. Proceedings PARLE'93, Munich (1993).
4. Rossainz, M., Capel M.: A Parallel Programming Methodology using Communication Patterns named HLPCS or Composition of Parallel Object. Proceedings of 20TH European Modeling & Simulation Symposium. Campora S. Giovanni. Italy (2008).
5. Rossainz, M., Capel M.: Design and implementation of communication patterns using parallel objects. International Journal of Simulation and Process Modelling. Volume 12, No.1, Pp: 69-91. ISSN: 1740-2131, (2017).
6. Robbins, K. A., Robbins S.: UNIX Programación Práctica. Guía para la concurrencia, la comunicación y los multihilos. Prentice Hall. (1999).
7. Roosta, Seyed: Parallel Processing and Parallel Algorithms. Theory and Computation. Springer (1999).
8. Wilkinson B., Allen M: Parallel Programming. Techniques and Applications Using Networked Workstations and Parallel Computers". Prentice-Hall. U.S.A. (1999).
9. Corradi A., Leonardi L.: PO Constraints as tools to synchronize active objects. Pp: 42-53. Journal Object Oriented Programming 10. (1991).
10. Danelutto, M.; Orlando, S; et al.: Parallel Programming Models Based on Restricted Computation Structure Approach. Technical Report-Dpt. Informatica. Universitá de Pisa (1999).
11. Rossainz M., Pineda I., Domínguez P.: Análisis y Definición del Modelo de las Composiciones Paralelas de Alto Nivel llamadas CPANs. Modelos Matemáticos y TIC: Teoría y Aplicaciones. Dirección de Fomento Editorial. ISBN 987-607-487-834-9. Pp. 1-19. México. (2014).
12. Almeida F., Giménez D., Mantas J.M., Vidal A.M.: Introducción a la Programación Paralela". Paraninfo CENAGE Learning. (2008).
13. Blelloch, Guy E.: Programming Parallel Algorithms. Communications of the ACM, Vol. 39, No. 3 (1996)
14. Butenhof, D. R. "Programming with POSIX® Threads". Addison Wesley. 1997.
15. Arjomandi E., O´Farrell W.G., Wilson G. V. "An Object-Oriented Communication Mechanism for Parallel Systems". Conference on Object-oriented Technologies. Toronto, Ontario, Canada, 1996. USENIX http://www.usenix.org.